Alexandria National University
Faculty of Computer and Data Science
Program :Cyber Security
Professor name: Dr. reem

منة الله اشرف على محمد 2205252

Report:

1. Objective of the Code and the Experiment

In this code, I build a simple experiment using a **Graph Neural Network (GraphSAGE)** to do **node classification** on a small graph.

The main idea is that I have a group of users in a network (for example, a social network). Some of them are **benign (normal/safe users)** and some are **malicious (harmful users)**, and I want to train a model that can predict whether each user is benign or malicious based on:

- The **features** of each user
- The **connections (edges)** between users in the graph

I use **PyTorch Geometric** to take advantage of the ready-made **GraphSAGE layers**.

---

2. Preparing the Graph Data

2.1 Node Features x

First, I define the tensor x, which represents the **features of each node** in the graph:

```
x = torch.tensor(
  [
    [1.0, 0.0],  # Node 0 (benign)
    [1.0, 0.0],  # Node 1 (benign)
    [1.0, 0.0],  # Node 2 (benign)
    [0.0, 1.0],  # Node 3 (malicious)
    [0.0, 1.0],  # Node 4 (malicious)
```

```
        [0.0, 1.0]   # Node 5 (malicious)
    ],
    dtype=torch.float,
)
```

- I have **6 nodes** (from 0 to 5).
- Each node has **2 features**:
  - [1, 0] = benign user
  - [0, 1] = malicious user

So here I'm giving the model a simple **one-hot encoding** that tells it there are only two types of nodes, benign and malicious, and each type has its own fixed feature vector.

2.2 Defining the Edges edge_index

Then I define edge_index, which represents the **connections (edges)** between the nodes:

```
edge_index = (
    torch.tensor(
        [
            [0, 1],
            [1, 0],
            [1, 2],
            [2, 1],
            [0, 2],
            [2, 0],
```

```
            [3, 4],
            [4, 3],
            [4, 5],
            [5, 4],
            [3, 5],
            [5, 3],
            [2, 3],
            [3, 2],
        ],
        dtype=torch.long,
    )
    .t()
    .contiguous()
)
```

I am treating the graph as **undirected**, so for every edge (u, v) I also add (v, u):

- Nodes **0, 1, 2** form a **complete triangle** → this is the **benign group** that is well connected internally.
- Nodes **3, 4, 5** form another triangle → this is the **malicious group**.
- Finally, I add **one edge between 2 and 3** to create a **connection between the benign world and the malicious world** (a benign user connected to a malicious user).

This makes the scenario a bit more interesting, because the model sees that **Node 2 (benign)** is connected to nodes in the malicious cluster.

2.3 Defining the Labels y

Next, I define the labels (the **true class** for each node):

y = torch.tensor([0, 0, 0, 1, 1, 1], dtype=torch.long)

- 0 = benign
- 1 = malicious

So:

- Nodes 0, 1, 2 → benign → label = 0
- Nodes 3, 4, 5 → malicious → label = 1

2.4 Packing Everything into a Data Object

PyTorch Geometric expects the data to be put into a Data object:

```
data = Data(x=x, edge_index=edge_index, y=y)
```

Here:

- data.x = node features
- data.edge_index = edges
- data.y = ground-truth labels

This data object is what I pass to the model during **training** and **evaluation**.

---

3. Designing the GraphSAGE Model

3.1 Defining the GraphSAGENet Class

I build a simple model with **two GraphSAGE layers**:

class GraphSAGENet(torch.nn.Module):

```
    def __init__(self, in_channels, hidden_channels, out_channels):
```

```
        super(GraphSAGENet, self).__init__()
        self.conv1 = SAGEConv(in_channels, hidden_channels)
        self.conv2 = SAGEConv(hidden_channels, out_channels)

    def forward(self, x, edge_index):
        x = self.conv1(x, edge_index)
        x = F.relu(x)
        x = self.conv2(x, edge_index)
        return F.log_softmax(x, dim=1)
```

Architecture explanation:

- in_channels = 2 → because each node has 2 features.
- hidden_channels = 4 → I chose a 4-dimensional hidden embedding.
- out_channels = 2 → I have 2 classes (benign and malicious), so the model outputs 2 scores (logits) per node.

3.2 What Happens Inside forward (Step by Step)

1. self.conv1(x, edge_index)
    - GraphSAGE takes the node features and:
        - Reads each node's own features.
        - Aggregates information from its neighbors based on edge_index.
        - Produces a new **embedding** for each node that combines its own features + the neighbors' features.
2. F.relu(x)

- I apply the **ReLU activation function** to add non-linearity and improve the expressive power of the model.

3. self.conv2(x, edge_index)
   - I apply a second GraphSAGE layer on top of the first-layer embeddings.
   - The output now has shape [num_nodes, out_channels], i.e., for each node I get 2 values (one for each class).
4. F.log_softmax(x, dim=1)
   - I convert the logits into **log-probabilities** over the two classes.
   - I use log_softmax because later, in the loss function, I use F.nll_loss, which expects log-probabilities.

---

4. Training Phase (Training Loop)

4.1 Creating the Model and the Optimizer

model = GraphSAGENet(in_channels=2, hidden_channels=4, out_channels=2)

optimizer = torch.optim.Adam(model.parameters(), lr=0.01)

- I choose the **Adam** optimizer because it usually works well out of the box without much tuning.
- The learning rate lr = 0.01 is a reasonable value for such a small and simple experiment.

4.2 The Training Loop

```
model.train()

for epoch in range(50):

   optimizer.zero_grad()

   out = model(data.x, data.edge_index)
```

```
loss = F.nll_loss(out, data.y)
loss.backward()
optimizer.step()
```

At each epoch, the following steps happen:

1. optimizer.zero_grad()
    - I reset the gradients from the previous iteration.
2. out = model(data.x, data.edge_index)
    - I run the **forward pass**:
        - The model receives the features and the edges.
        - It outputs out, which are the log-probabilities for each node over the 2 classes.
3. loss = F.nll_loss(out, data.y)
    - I compute the **loss** between the model predictions and the true labels y.
    - Since the model outputs log-softmax, nll_loss is the correct choice here.
4. loss.backward()
    - I perform **backpropagation**, computing gradients for all model parameters.
5. optimizer.step()
    - I update the model weights using the gradients and the learning rate.

I train the model for **50 epochs**, which is more than enough for such a tiny graph with only 6 nodes.

---

5. Evaluation and Reading the Results

After training, I switch the model to evaluation mode:

```
model.eval()

pred = model(data.x, data.edge_index).argmax(dim=1)

print("Predicted labels:", pred.tolist())
```

- model.eval()
  - This turns off training-specific behaviors (like Dropout or BatchNorm if they existed), so evaluation is stable.
- model(data.x, data.edge_index)
  - I run a forward pass again to get the log-probabilities for each node.
- .argmax(dim=1)
  - I take the class with the highest log-probability for each node:
    - 0 → predicted as benign
    - 1 → predicted as malicious

If training goes well, I expect to see something like:

Predicted labels: [0, 0, 0, 1, 1, 1]

This means the model has successfully learned to **separate the benign group from the malicious group** and classify all 6 nodes correctly based on both their features and their connections in the graph.